CS 241: Systems Programming
Lecture 15. Enums

Fall 2025

Prof. Stephen Checkoway

Process states

Process states

Every process in the system is in one of several states

Process states

Every process in the system is in one of several states

- Running/Runnable — Process is running on a CPU or able to run
- Interruptable sleep — Process is asleep but can be awakened via a signal
- Uninterruptable sleep — Process is asleep but will not wake for a signal
- Stopped — Process has been suspended (e.g., ctrl-Z)
- Zombie — Process has exited but is still in the process table until its parent uses the wait system call to “reap” it

Printing the process state

```
$ ps -e -o pid,state,command
```

This will print the process ID, process state, and command name of every process on the system

```
    PID S COMMAND
      1 S /sbin/init splash
      …
1156303 R sshd: mhogan@pts/0
1156310 S -bash
1156474 S /usr/libexec/tracker-store
1156493 R ps -e -o pid,state,command
```

Modeling the process state

Enums let us model a situation where a value is one of a set of possible values, called **variants**

```
/// Every process is in one of these possible states
enum ProcessState {
    /// Process is running on a CPU
    Running,

    /// Process is ready to be run
    Runnable,

    /// Process is asleep but can be awakened by a signal
    InterruptableSleep,

    /// Process is asleep but cannot be awakened by a signal
    UninterruptableSleep,

    /// Process is stopped
    Stopped,

    /// Process has died but hasn't yet been "reaped"
    Zombie,
}
```

Using an enum

```
let running = ProcessState::Running;
let stopped = ProcessState::Stopped;
```

In general, you name a variant as EnumName::VariantName

Match

We can implement methods for enums

```
impl ProcessState {
    fn is_asleep(&self) -> bool {




    }
}
```

Match

We can implement methods for enums

```
impl ProcessState {
    fn is_asleep(&self) -> bool {
        match self {
        }
    }
}
```

Match

We can implement methods for enums

```
impl ProcessState {
    fn is_asleep(&self) -> bool {
        match self {
            ProcessState::Running => false,
            ProcessState::Runnable => false,
            ProcessState::InterruptableSleep => true,
            ProcessState::UninterruptableSleep => true,
            ProcessState::Stopped => false,
            ProcessState::Zombie => false,
        }
    }
}
```

Match

We can implement methods for enums

```
impl ProcessState {
    fn is_asleep(&self) -> bool {
        match self {
            ProcessState::Running => false,
            ProcessState::Runnable => false,
            ProcessState::InterruptableSleep => true,
            ProcessState::UninterruptableSleep => true,
            ProcessState::Stopped => false,
            ProcessState::Zombie => false,
        }
    }
}
```

match statements must cover all variants

Calling methods on enums

```
fn main () {
    let running = ProcessState::Running;
    let stopped = ProcessState::Stopped;

    println!("{}", ProcessState::InterruptableSleep.is_asleep());
    println!("{}", running.is_asleep());
}
```

What will this code print?

```
fn main () {
    let running = ProcessState::Running;
    let stopped = ProcessState::Stopped;

    print!("{} ", ProcessState::InterruptableSleep.is_asleep());
    println!("{}", running.is_asleep());
}
```

A. true true

B. true false

C. false false

D. false true

E. This code will not compile

Match

```
impl ProcessState {
    fn is_asleep(&self) -> bool {
        match self {
            ProcessState::Running => false,
            ProcessState::Runnable => false,
            ProcessState::InterruptableSleep => true,
            ProcessState::UninterruptableSleep => true,
            ProcessState::Stopped => false,
            ProcessState::Zombie => false,
        }
    }
}
```

Match with wildcard _

```
impl ProcessState {
    fn is_asleep(&self) -> bool {
        match self {
            ProcessState::InterruptableSleep => true,
            ProcessState::UninterruptableSleep => true,
            _ => false,
        }
    }
}
```

matches everything

Enums with data

We can associate different (types and amounts of) data with each variant

```
enum Color {
    White,
    Black,
    Red,
    Green,
    Blue,
    Other(u8, u8, u8),
}

fn main() {
    let black: Color = Color::Black,
    let pink: Color = Color::Other(247, 98, 210);
}
```

Matching enums with data

```
fn main() {
    let color = Color::Other(200, 100, 22);

    match color {
        Color::White => println!("White"),
        Color::Black => println!("Black"),
        Color::Red => println!("Red"),
        Color::Green => println!("Green"),
        Color::Blue => println!("Blue"),


        }
    }
}
```

Matching enums with data

```
fn main() {
    let color = Color::Other(200, 100, 22);

    match color {
        Color::White => println!("White"),
        Color::Black => println!("Black"),
        Color::Red => println!("Red"),
        Color::Green => println!("Green"),
        Color::Blue => println!("Blue"),
        Color::Other(red, green, blue) => {
            println!("({red}, {green}, {blue})");
        }
    }
}
```

Matching enums with data

```
fn main() {
    let color = Color::Other(200, 100, 22);

    match color {
        Color::White => println!("White"),
        Color::Black => println!("Black"),
        Color::Red => println!("Red"),
        Color::Green => println!("Green"),
        Color::Blue => println!("Blue"),
        Color::Other(red, green, blue) => {
            println!("({red}, {green}, {blue})");
        }
    }
}
```

Can use a block
for any match case

Matching enums with data

```
fn main() {
    let color = Color::Other(200, 100, 22);

    match color {
        Color::White => println!("White"),
        Color::Black => println!("Black"),
        Color::Red => println!("Red"),
        Color::Green => println!("Green"),
        Color::Blue => println!("Blue"),
        Color::Other(red, green, blue) => {
            println!("({red}, {green}, {blue})");
        }
    }
}
```

Can use a block for any match case

Omit the comma after a block

Enums with named data

```
enum Color {
    Hsv {
        hue: u16,
        saturation: u8,
        value: u8,
    },
    Rgb {
        red: u8,
        green: u8,
        blue: u8
    },
    Cmyk {
        cyan: u8,
        magenta: u8,
        yellow: u8,
        black: u8,
    },
}
```

```
fn main() {
    let pink: Color = Color::Rgb {
        red: 247,
        green: 98,
        blue: 210
    };
    let dark_green: Color = Color::Hsv {
        hue: 111,
        saturation: 96,
        value: 51
    };
    let gray: Color = Color::Cmyk {
        cyan: 0,
        magenta: 0,
        yellow: 0,
        black: 25
    };
}
```

Variants aren’t separate types!

Variants aren’t separate types!

It’s important to recognize that an enum’s variants aren’t separate types

Variants aren’t separate types!

It’s important to recognize that an enum’s variants aren’t separate types

```
let invalid: Color::Rgb = Color::Rgb {
    red: 247,
    green: 98,
    blue: 210
};
```

Variants aren’t separate types!

It’s important to recognize that an enum’s variants aren’t separate types

```
let invalid: Color::Rgb = Color::Rgb {
    red: 247,
    green: 98,
    blue: 210
};
```

```
error[E0573]: expected type, found variant `Color::Rgb`
   --> enums.rs:243:14
    |
243 | let invalid: Color::Rgb = Color::Rgb {
    |              ^^^^^^^^^^
    |              |
    |              not a type
    |              help: try using the variant's enum: `Color`
```

We can match enums with named data by using a names for the fields. Which of the following is a correct match on the Color type with variants Hsv, Rgb, and Cmyk?

```
// A
match color {
    Color::Rgb { red, green, blue } => {
        println!("{red}, {green}, {blue}")
    }
    _ => ()
}

// B
match color {
    Color::Rgb { red, green, blue } => {
        println!("{red}, {green}, {blue}")
    }
}
```

```
// C
match color {
    Rgb { red, green, blue } => {
        println!("{red}, {green}, {blue}")
    }
    _ => ()
}

// D
match color {
    Rgb { red, green, blue } => {
        println!("{red}, {green}, {blue}")
    }
}

// E. More than one of the above.
```

Structs vs. enums

Structs and enums both group related data

Structs are useful when each instance always has multiple, related values

Enums are useful when you sometimes have some data and others times have other data

Every process has some data associated with it. It has a process state and a user ID (uid) and a group ID (gid) among other data. Which of these definitions of Process should you use to model this?

```
struct Process {
    state: ProcessState,
    uid: u32,
    gid: u32,
}

enum Process {
    State(ProcessState),
    Uid(u32),
    Gid(u32),
}
```

A. enum

B. struct

C. Either struct or enum (both work)

D. Neither struct nor enum

Debug representation, Clone

Like with structs, we can (and probably should) derive Debug and Clone

```
/// Every process is in one of these possible states
#[derive(Debug, Clone)]
enum ProcessState {
    /// Process is running on a CPU
    Running,
    …
}
```

Comparing enum values with ==

```
fn main () {
    let state = ProcessState::Running;

    if state == ProcessState::Stopped {
        todo!()
    }
}
```

Comparing enum values with ==

```
fn main () {
    let state = ProcessState::Running;

    if state == ProcessState::Stopped {
        todo!()
    }
}
error[E0369]: binary operation `==` cannot be applied to type `ProcessState`
  --> enums.rs:52:14
   |
52 |     if state == ProcessState::Stopped {
   |        ----- ^^ ------------------- ProcessState
   |        |
   |        ProcessState
   |
note: an implementation of `PartialEq` might be missing for `ProcessState`
  --> enums.rs:6:1
   |
6  | enum ProcessState {
   | ^^^^^^^^^^^^^^^^^ must implement `PartialEq`
help: consider annotating `ProcessState` with `#[derive(PartialEq)]`
```

Derive PartialEq and Eq

```
/// Every process is in one of these possible states
#[derive(Debug, Clone, PartialEq, Eq)]
enum ProcessState {
    /// Process is running on a CPU
    Running,
    …
}
```

PartialEq gives us access to == and !=.

Eq adds nothing else but informs the compiler that ProcessStates are equal to themselves

Option

A built-in enum that is either a `None` or a `Some(x)` for some value `x`

```
enum Option<T> {
    Some(T),
    None,
}
```

Option

A built-in enum that is either a `None` or a `Some(x)` for some value `x`

```
enum Option<T> {
    Some(T),
    None,
}

let x: Option<String> = None;
let y: Option<u32> = Some(9123474);
```

The <T> is a type parameter. We have different types of `Option` depending on T

Option models the situation where a value may be absent

Uses of Option:

- Implementing optional command line arguments using clap

```
/// Print LINES lines of each of the specified files
#[arg(short = 'n', long)]
lines: Option<usize>,
```

- Searching for a value in a collection

```
let s = String::from(…);
let pos: Option<usize> = s.find('🤓');
```

Result

A built-in enum that is either Ok(x) or Err(y) for some values x and y

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Many methods in the Rust standard library return a Result

Many methods in the Rust standard library return a Result

All of the functions that perform input/output return a `std::io::Result<T>`

Many methods in the Rust standard library return a Result

All of the functions that perform input/output return a `std::io::Result<T>`

`std::io::Result<T>` is a type alias for `Result<T, std::io::Error>`

- This is a normal `Result` with a specialized error type `std::io::Error`

Many methods in the Rust standard library return a Result

All of the functions that perform input/output return a `std::io::Result<T>`

`std::io::Result<T>` is a type alias for `Result<T, std::io::Error>`

- This is a normal `Result` with a specialized error type `std::io::Error`

Examples

- Opening a file with `File::open(path)` returns an `io::Result<File>`
- Creating a file with `File::create(path)` returns an `io::Result<File>`
- `.read()` on a file returns an `io::Result<usize>` where the size is the number of bytes read
- `.write_all()` on a file returns an `io::Result<()>` where the `Ok(())` indicates success but carries no additional data

Aside: Name variants without enum name: use statement

We're allowed to use the Some(x), None, Ok(x), and Err(x) variants of Option and Result without specifying the enum itself

- I.e., we don't have to say Option::Some(x)

We can bring the names of the variants of our enums into scope with a use statement

```
#[derive(Debug, Clone)]
enum Either<T1, T2> {
    Left(T1),
    Right(T2),
}

fn main() {
    use Either::*;

    let x: Either<i32, &str> = Left(5);
    let y: Either<i32, &str> = Right("Hello");
    println!("{x:?}\n{y:?}");
}
```

Of the following string methods, one returns an Option and one returns a Result. Which do you think returns what?

`String::find(pat: P)` – on success, returns the byte index of the first character of the string slice that matches the pattern pat

`String::parse<F>()` – on success, parses this string slice into another type

	`String::find(pat: P)`	`String::parse<F>()`
A	**Option**	**Option**
B	**Option**	**Result**
C	**Result**	**Option**
D	**Result**	**Result**

Propagating errors explicitly

```
use std::fs::File;
use std::io::{self, BufRead, BufReader};

fn read_first_line(path: &str) -> io::Result<String> {
    let tmp = File::open(path);

    if tmp.is_err() {
        return tmp;
    }
    let file = tmp.unwrap();

    let mut reader = BufReader::new(file);
    let mut line = String::new();

    let tmp = reader.read_line(&mut line);
    if tmp.is_err() {
        return tmp;
    }
    Ok(line)
}
```

Propagating errors using ?

```rust
use std::fs::File;
use std::io::{self, BufRead, BufReader};

fn read_first_line(path: &str) -> io::Result<String> {
    let file = File::open(path)?; // Returns any errors
    let mut reader = BufReader::new(file);
    let mut line = String::new();

    reader.read_line(&mut line)?; // Returns any errors
    Ok(line)
}
```

Using match to handle Results

```
fn main() {
    let path = "file.txt";
    let result = read_first_line(path);
    match result {
        Ok(line) => {
            println!("First line: {line}");
        }
        Err(err) => {
            // Write the error to stderr
            eprintln!("{path}: {err}");
        }
    }
}
```

Generic Result type

```
type Result<T> = std::result::Result<T, Box<dyn std::error::Error>>;
```

The error type is a `Box` holding any type that implements the `Error` trait

- All of the standard library error types (like `std::io::Error`) implement `Error`

If `result` is an `Err(err)`, then `result?` will try to convert `err` into the correct error type to be returned from the function

- Any type that implements `Error` can be turned into a `Box<dyn Error>`
- A `String` can be turned into a `Box<dyn Error>`

Match and ownership

If an enum has data, then matching an instance of the enum will move the data

Match and ownership

If an enum has data, then matching an instance of the enum will move the data

```
fn main() {
    let opt: Option<String> = Some(String::from("owned"));

    match opt {
        None => (),
        Some(s) => println!("{s}"), // Moves out of the opt
    }
    println!("{opt:?}");
}
```

Error message

```
error[E0382]: borrow of partially moved value: `opt`
   --> enums.rs:179:15
    |
177 |             Some(s) => println!("{s}"), // Moves out of the opt
    |                  - value partially moved here
178 |         }
179 |         println!("{opt:?}");
    |                   ^^^^^^^ value borrowed here after partial move
```

Two solutions

Two solutions

1. Match on &opt instead which gives a reference to the inner data

```
match &opt {
    None => (),
    Some(s) => println!("{s}"), // s is a reference
}
```

Two solutions

1. Match on &opt instead which gives a reference to the inner data

```
match &opt {
    None => (),
    Some(s) => println!("{s}"), // s is a reference
}
```

2. Use the `ref` keyword to indicate the pattern should bind a reference to the data

```
match opt {
    None => (),
    Some(ref s) => println!("{s}"), // s is a reference
}
```

if let (a match alternative)

if let (a match alternative)

In many cases, you only care if an enum is a particular variant

```
    match s.find("tr") {
        Some(idx) => {
            println!("Substring 'tr' found at index {idx}");
        }
        _ => {
            println!("Substring 'tr' not found");
        }
    }
```

if let (a match alternative)

In many cases, you only care if an enum is a particular variant

```
    match s.find("tr") {
        Some(idx) => {
            println!("Substring 'tr' found at index {idx}");
        }
        _ => {
            println!("Substring 'tr' not found");
        }
    }
```

can be written more simply using `if let`

if let (a match alternative)

In many cases, you only care if an enum is a particular variant

```
    match s.find("tr") {
        Some(idx) => {
            println!("Substring 'tr' found at index {idx}");
        }
        _ => {
            println!("Substring 'tr' not found");
        }
    }
```

can be written more simply using `if let`

```
    if let Some(idx) = s.find("tr") {
        println!("Substring 'tr' found at index {idx}");
    } else {
        println!("Substring 'tr' not found");
    }
```

if let (a match alternative)

In many cases, you only care if an enum is a particular variant

```
    match s.find("tr") {
        Some(idx) => {
            println!("Substring 'tr' found at index {idx}");
        }
        _ => {
            println!("Substring 'tr' not found");
        }
    }
```

can be written more simply using `if let`

```
    if let Some(idx) = s.find("tr") {
        println!("Substring 'tr' found at index {idx}");
    } else {
        println!("Substring 'tr' not found");
    }
```

There's a similar `while let pattern = expr { }`